Verification of COMDES-II Systems Using UPPAAL with Model Transformation

Xu, Ke; Pettersson, Paul; Sierszecki, Krzysztof; Angelov, Christo K.

Published in:
Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA'2008

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

Verification of COMDES-II Systems Using UPPAAL with Model Transformation*

Xu Ke†, Paul Pettersson‡, Krzysztof Sierszecki†,Christo Angelov†
†Mads Clausen Institute for Product Innovation, University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
{xuke, ksi, angelov}@mci.sdu.dk
‡School of Innovation, Design and Engineering, Mälardalen University
Högskoleplan 1, Box 883, 72123, Västerås, Sweden
paul.pettersson@mdh.se

Abstract

COMDES-II is a component-based software framework intended for model-integrated development of embedded control systems with hard real-time constraints. It provides various kinds of component models to address critical domain-specific issues, such as real-time concurrency and communication in a timed multitasking environment, modal continuous operation combining reactive control behavior with continuous data processing, etc., by following the principle of separation-of-concerns. In the paper we present a transformational approach to the formal verification of both timing and reactive behaviors of COMDES-II systems using UPPAAL, based on a semantic anchoring methodology. The proposed approach adopts UPPAAL timed automata as the semantic units, to which different behavioral concerns of COMDES-II are anchored, such that a COMDES-II system can be precisely specified in UPPAAL, and verified against a set of desired requirements with the preservation of system original operation semantics.

1. Introduction

Recently emerging concepts and techniques, such as *model-integrated development* (MID) and *component-based design* (CBD) have considerable implications for the efficient development of reliable embedded software systems [1]. On one hand, MID advocates a domain-specific model-driven approach to facilitate the development of embedded software, by equipping developers with a domain-specific modeling language (DSML) embodying the modeling concepts, constraints and assumptions of application domains. On the other hand, CBD can be regarded as one of the most suitable design paradigms for MID, due to the considerable benefits brought by reusability of components and higher-level of abstraction. Moreover, from a software engineering point of view, CBD is also an effective way to bridge the gap between conceptual system design models and concrete system implementations [2], provided that an automatic code generation technique has been developed.

COMDES-II is a component-based software framework adopting MID as the methodological basis to develop distributed control systems with hard real-time constraints [3, 4]. In order to achieve this objective, COMDES-II provides various kinds of component models to address crucial domain-specific issues, such as system concurrency, real-time operation, sequential control behavior combined with continuous data processing, etc., by following a *separation-of-concerns* approach [4]. A meta-modeling process formally defines the syntax and static semantics of framework component models [1], however, specification and verification of composed component behavior are still a challenging problem.

Semantic anchoring [1, 5] is a promising approach to the transformational specification and verification of system dynamic behavior, by relying on *semantic units* (such as finite state machines, timed automata etc.) with well-defined operational semantics and tool support. Briefly, the elements and their relationships in a DSML can be equivalently transformed onto their counterparts in an executable semantic unit with well-defined behavior. The resulting model can be subsequently validated and verified using supporting toolsets, while preserving the original system operational semantics. This transformation process from original DSML to the corresponding semantic unit is referred to as *semantic anchoring*, as shown in Figure 1.

We choose timed automata in UPPAAL as the semantic unit, and this paper presents the concrete process of developing such a transformational approach to specify and verify the behavior of COMDES-II systems via semantic anchoring. The structure of the paper follows a logical sequence: Section 2 and Section 3 provide an overview of COMDES-II component models and timed automata in UPPAAL respectively, which would give a general perspective on semantic gaps between the two kinds of systems. Section 4 describes in details how the semantic differences are bridged via an extensive model transformation process in all behavioral aspects. Finally, the concluding section summarizes the features of our work and their implications.

*This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

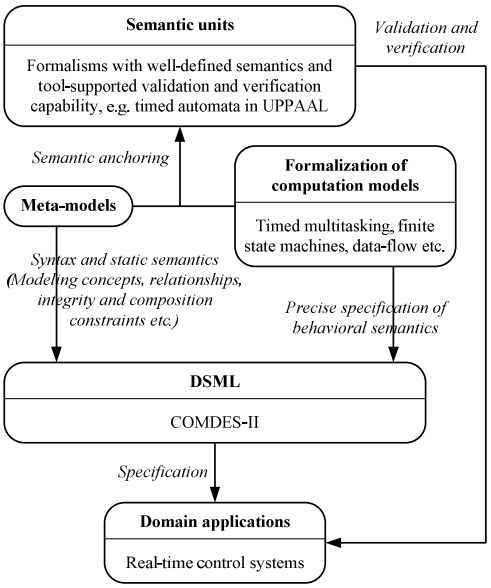

Figure 1. Semantic anchoring of CODMES-II

Contribution. This paper describes a research effort as a continuation of the work which has been presented in [4]. The main contribution is that we propose a transformational approach that supports the efficient verification of *schedulability* and *reactive functionality* for *preemptive timed multitasking* systems (which is the primary behavioral semantics of COMDES-II) by using UPPAAL. The extensive semantic-preserving model transformation performed at the meta-level guarantees that predictable system operation and deterministic task sequential behavior – as defined in COMDES-II – are precisely captured in transformed UPPAAL models, resulting in much reduced verification state space and minimized occurrence of spurious counter-examples.

2. System Modeling in COMDES-II

As a component-based design framework intended for real-time control systems, COMDES-II takes into account both the architectural and behavioral characteristics of the targeted domain during a system development process.

With respect to architecture, COMDES-II employs a hierarchical model to specify a system structure: at the system level a control application is conceived as a network of communicating *actors* that interact transparently with each other by exchanging labeled state messages (*signals*), following a producer-consumer communication protocol. Actors are active components, which are composed of multiple *I/O drivers* and a single *actor task*. The I/O drivers are responsible for acquiring or generating signals from/to network or physical units, while the actor task processes the input signals to perform the required functionality which is specified by a composition of different *function block instances*. Function block instances are instantiations of reusable and reconfigurable function block *types*, which can be categorized into four function block *kinds* (meta-types): basic, composite, modal as well as state machine function blocks. A detailed description of the COMDES-II systems architecture and function block models is out of the scope of this paper and we refer the interested readers to [3, 4].

In another aspect, COMDES-II system behavior is of particular interest to our model transformation and analysis effort. In COMDES-II, a separation-of-concerns approach is extensively applied to help clearly model different behavioral concerns, such that scheduling, timing and functional behaviors are separated from each other. Specifically, scheduling and real-time issues are specified with respect to actors, while functional behavior can be modeled by the composition of various kinds of function block instances contained within actor tasks.

The scheduling policy of real-time operation of actors follows a *fixed-priority timed multitasking* (TM) model of computation [6], in which actors are activated by either periodic or aperiodic events, and execute preemptively according to their assigned priorities with non-blocking *read-do-write* semantics, as follows:

- Upon activation, input drivers of the activated actor will be invoked (*read*) in logically zero time to acquire all input signals which are latched throughout the whole actor execution.
- The activated actor task will process (*do*) exactly once the input data as long as it becomes the highest priority task among all released/preempted tasks in the processor. The input signals provide all the information for an actor task to determine its current state and the associated computation results.
- The computed control results will then be buffered into output drivers that can be atomically executed to generate (*write*) output signals when the corresponding actor deadline expires. If the deadline of an actor is not specified (i.e. deadline = 0), the actor output drivers will be immediately executed when the actor task finishes its execution.

This non-blocking operation pattern is accomplished via a time-triggered scheduler that controls the timed I/O activities and *split-phase* execution of actor tasks and drivers over discrete-time (see Figure 2), resulting in a stepwise progress of actor execution time represented as multiples of a basic timing unit (i.e. the period of scheduler activation). Split-phase execution provides for predictable actor operation in the sense that the output signals of a specific actor are made available to other actors at its explicitly specified deadline instant, as long as all actor tasks are schedulable.

The four kinds of function blocks (FBs) defined in COMDES-II are pure functional components implementing concrete computation or control algorithms to fulfill different kinds of functionalities for host actors, as summarized in Table 1.

In particular, basic and composite FBs are used to model the data-flow computation process executing in a single

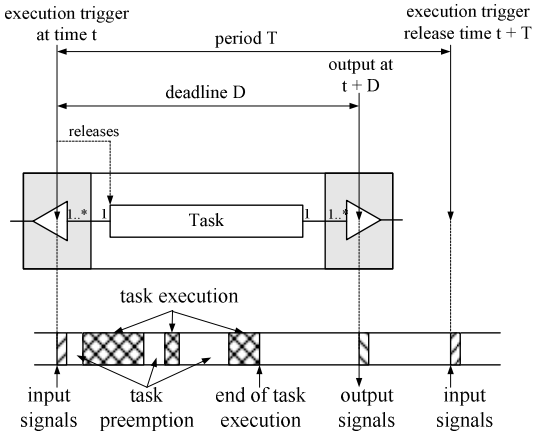

Figure 2. Split-phase execution of actors under timed multitasking

mode of operation, while state machine FBs (SMFBs) and modal FBs (MFBs) are jointly used to specify the system reactive behavior (control-flow) combined with modal data-flow computations, as shown in Figure 3. In the latter computation model, a modal FB has a number of operation modes (states) and each mode contains a function block diagram (data-flow process) representing the control action to be performed. The selection of executing state is decided by the currently active *state* information provided from the supervisory state machine FB, whereas the enabledness of executing state is determined by the *state_updated* value, i.e. the control action should only be performed when a state transition has actually taken place in the corresponding supervisory state machine FB. Since the system reactive behavior is particularly of our analysis interest, we will introduce the state machine FB in more detail and refer the interested readers to [4] for more information about the other kinds of FBs.

Table 1. Functionalities of function blocks

Functionalities	Function blocks
Continuous control behavior (*data-flow*)	Basic FBs and composite FBs
Reactive control behavior (*control-flow*)	State machine FBs
Hybrid control behavior (*modal continuous control*)	Composition of state machine FBs and modal FBs

A COMDES-II state machine FB F is defined as a tuple:

$$F_{<h>} = (I, O, M, f)$$

where:

- I is a finite set of binary event/guard input signals $\{i_1, i_2, \ldots, i_n | i : Boolean\}$ that will be used by the internal state machine to determine its current state

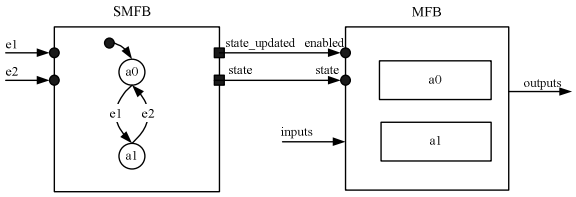

Figure 3. Interaction between a state machine FB (SMFB) and a modal FB (MFB)

- O is a finite set of output signals containing exactly two elements: $\{state, state\_updated | state : Integer, state\_updated : Boolean\}$. Here, $state$ represents the currently active state of F, and $state\_updated$ is set to $true$ if a state transition has happened, otherwise it is set to $false$. These two output signals are used to supervise the execution of corresponding modal FBs, as shown in Figure 3
- M is a signal-driven state machine model [4] specifying the internal state machine: $M = (S, s_0, T)$, where:
  - S is a finite set of states
  - s_0 represents the initial state, $s_0 \in S$
  - T is a finite set of transitions denoted as $s \longrightarrow_{(e,g,o)} s'$ and $s, s' \in S$, where e stands for an event signal, g denotes a guard defined in terms of input signals, o is the transition order starting from 1 to indicate the importance of transitions. Transitions are effected by transition *triggers* that are defined as event-guard combinations. When multiple triggers associated with outgoing transitions from a state s are simultaneously evaluated as true, transition orders can be used to fire the most important state transition that deterministically leads to a current state s', which is an appealing property to safety-critical control systems
- f is a function interpreting the control logics specified by the internal state machine of F to fufill the designated sequential control behavior
- $h : Boolean$ is an attribute of F indicating if the state machine is historic or not

For a better understanding, an example of a state machine FB called *SMFB_1* is shown in Figure 4, whose internal structure conforms to the definition given above. The *SMFB_1* contains three binary event inputs *e1*, *e2* and *e3*, a signal-driven state machine model, and exactly two outputs: *state* and *state_updated*. The internal state machine includes two states *s1* (initial state) and *s2*, three state transitions that are labeled by events and transition orders. When the host actor is activated and then *SMFB_1* is executed, its internal state machine will parse the binary input event signals, determine current state and then update the two outputs accordingly.

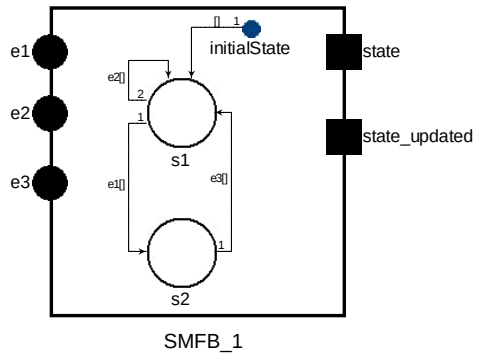

Figure 4. An example state machine FB *SMFB_1*

3. Timed automata in UPPAAL

The theory of timed automata has proven to be useful for specification and verification of real-time systems. In this section we briefly review the basic definitions needed in this paper. We refer the reader to [7] for a more thorough description of the timed automata used in the UPPAAL tool [8].

Assume a finite set of real-valued variables C standing for clocks, and a finite set of actions Act. Let $B(C)$ denote the set of Boolean combinations of clock constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in C$ and n is a natural number. Syntactically, a timed automaton A is a tuple $< N, l_0, E, I >$ where: N is a finite set of locations, $l_0 \in N$ is the initial location, $E \in N \times B(C) \times Act \times 2^C \times N$ is the set of edges, and $I : N \rightarrow B(C)$ assigns invariant clock constraints to locations.

The semantics of a timed automaton is a timed transition system with states of the form $< l, u >$, where $l \in N$ and u is a clock assignment assigning all clocks in C to a non-negative real-number. Transitions are defined by the two rules:

- (discrete transitions) $< l, u > \longrightarrow^a < l', u' >$ if $< l, g, a, r, l' > \in E$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$
- (delay transitions) $< l, u > \longrightarrow^d < l, u \oplus d >$ if $u \in I(l)$ and $(u \oplus d) \in I(l)$ for a non-negative real number d

where $u \oplus d$ denotes the clock assignment which maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ is the clock assignment u with each clock in r to be zero.

A network of automata is a finite set of *automata processes* composed in parallel with a CCS-like parallel composition operator [9]. For a network of processes with the timed automata $A_1, \ldots, A_n$ the intuitive meaning is similar to the CCS parallel composition of $A_1, \ldots, A_n$ with all actions being restricted, that is, $(A_1 | \ldots | A_n) \setminus Act$. Thus, an edge labeled with action a must synchronize with an edge labeled with an action complementary to a, whereas edges with the silent τ action are internal, so they do not synchronize. In UPPAAL '?' and '!' are used to represent complementary actions, so $a?$ and $a!$ are considered complementary and can synchronize. The silent τ action is represented in UPPAAL by no synchronization action (i.e., an edge with an empty synchronization action).

Finally, we note that the flavor of timed automata used in the UPPAAL tool is extended with data variables with finite domains, including Booleans and finite domain Integers, as well as records and (multidimensional) arrays of data variables, action channels, and clocks. In UPPAAL it is also possible to declare functions defined in a C-like programming language that can be sequentially composed with the resets r of edges. The programming language allows for branching with *if/then/else* statements, *for*, *while* and *do/while* loops, and a *return* statement. We refer readers to the online help available on the UPPAAL homepage[1] for more information about this feature.

4. Transformation from COMDES-II to UPPAAL

The discussion made in the previous two sections has shown that *processes* and *timed automata* in UPPAAL may act as the basic architectural elements, to which *actors* and *state machine FBs* in COMDES-II can be anchored. However, the scheduling policy of actors and the operational semantics of state machine FBs differ from their counterparts in UPPAAL in all aspects, as listed in Table 2, which requires an extensive model transformation be performed at the meta-level to bridge the semantic gaps between the two languages. In Section 4.1 and Section 4.2 we will show how these gaps are bridged individually – by following a separation of behavioral concerns approach.

Table 2. Behavioral differences between COMDES-II and UPPAAL

Behavioral aspects	COMDES-II	UPPAAL
Concurrency	Fixed-priority preemptive scheduling of actors with non-blocking read-do-write semantics	Interleaving parallelism of timed automata processes
Time	Execution of timed I/O activities over discrete real-time	Continuous model-time
Reactive behavior	State machine FB as introduced in Section 2	Timed automata as introduced in Section 3

4.1. Transformation of concurrency and time

The preemptive timed multitasking (TM) scheduling policy of actors in COMDES-II is principally different from the interleaving parallelism of UPPAAL processes as de-

[1] The UPPAAL home page is located at www.uppaal.com.

fined in CCS. The key factor to overcome concurrency and timing differences between the two kinds of systems is to identify how to model the discrete-time scheduler that controls the actor execution status in UPPAAL. In order to achieve this objective, the following four modeling procedures should be accomplished successively:

- Finding out a way to model actors and represent their execution information
- Modeling actor interaction following a state message producer-consumer communication protocol
- Establishing a method to manage the non-blocking read-do-write concurrency of actors, which execute preemptively in a discrete real-time domain
- Modeling the discrete-time scheduler based on the previous three steps

Our solutions will be presented step-by-step in the following subsections.

4.1.1. Modeling actors in UPPAAL. The COMDES-II actor model has been briefly introduced in Section 2, from which we can see that the *read* (input drivers), *do* (actor task) and *write* (output drivers) actions of a specific actor are performed in an ordered sequence within split timing phases (see Figure 2). Hence, from a temporal point of view, it is natural to separately model these three kinds of actor behaviors using different software artifacts, so that they can be easily controlled by the time-triggered scheduler.

In UPPAAL, actor tasks are specified by the corresponding task control blocks containing all the information needed for scheduling task execution, as defined below:

```
typedef struct {
        int[0,4] status;
        meta int period;
        meta int executionTime;
        meta int deadline;
        meta int mode;
        bool modeUpdated;
        int timeSinceReleased;
        int computationTimer;
} TTask;
```

In which `status` is a bounded-value integer ([0, 4]) denoting the execution status of a given task. A task could be in `READY` (0), `ACTIVE` (1), `COMPUTED` (2), `FINISHED` (3) or `ERROR` (4) status which are determined and updated by the system scheduler. For a better understanding, Figure 5 conceptually illustrates the status transition graph of a specific task over discrete-time, whose concrete meaning and determination strategy will be subsequently explained in Section 4.1.3. The three integers `period`, `executionTime` and `deadline` represent the execution period, worst-case execution time and deadline of a specific task, and their values remain unchanged during system execution. Therefore they are declared as *meta* integers whose values are used for task execution and scheduling, but are not recorded in the verification state space. Another meta integer – `mode` – indicates the currently active state (mode) of the task state machine (e.g. product_ready, pre_processing, etc.), and the Boolean variable `modeUpdated` is used to denote if a state transition has happened or not in the current cycle of execution. These two variables encode the interaction between a state machine FB and a modal FB, and are used by output drivers to control the generation of output signals associated with the current state at the task deadline instant, if a state transition has taken place.

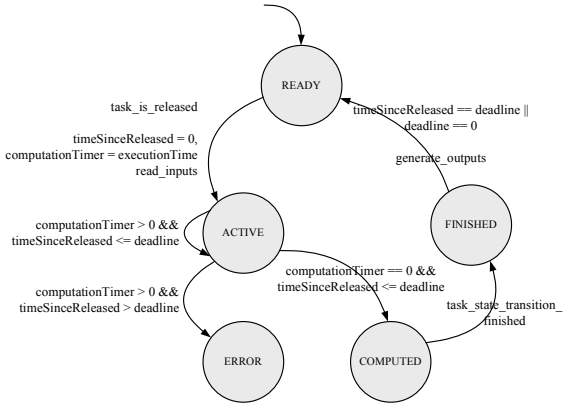

Figure 5. Status transition graph of actor tasks

The integer `timeSinceReleased` represents the discrete-time that has elapsed since a specific task is released: upon release of the given task, this variable is reset (see Figure 5), and is subsequently incremented each time the scheduler is executed. The integer variable `computationTimer` is a timer used to record the time left for a task computation. This variable is set to the value of `executionTime` when a task is released (see Figure 5), and count down if the task is the highest-priority `ACTIVE` task each time the scheduler is executed. The value of this timer is used to determine the task status, as well as for schedulability analysis. A detailed explanation is given in Section 4.1.3.

The task control block `TTask` can be instantiated into an array of tasks to specify their execution and scheduling information, and the array index (starting from 1) corresponds to the priority of each task: the higher index, the higher priority.

Actor I/O drivers will be implemented within two functions in UPPAAL: `taskInputDrivers(int taskID)` and `taskOutputDrivers(int taskID)`. These two functions are application-dependent, and when a specific actor i is released or its deadline expires, they (`taskInputDrivers(i)` and `taskOutputDrivers(i)`) will be invoked and executed atomically to exchange information with other actors, as described in the next section.

4.1.2. Modeling actor interaction in UPPAAL. Communication between actors in COMDES-II follows a producer-

consumer protocol with signal-based non-blocking semantics. Signals are labeled state messages containing process data, while in UPPAAL a hand-shaking interaction mechanism is adopted to primarily synchronize the actions between automata processes as defined in CCS parallelism, and no data is exchanged between processes.

A natural way for solving this problem is to model the COMDES-II communication mechanism in UPPAAL through shared variables and data structures. Information between processes is exchanged by updating and reading from these global resources, where the data race problem is resolved by the following COMDES-II semantics:

- When multiple actors are released, or their deadlines expire simultaneously, the corresponding I/O functions will be invoked and executed sequentially according to the order of actor priorities
- If the deadline of actor i expires at the same instant as actor j is released ($i \neq j$), then the output action `taskOutputDrivers(i)` of actor i will be executed by preceding the input action `taskInputDrivers(j)` of actor j, regardless the order of i and j. This rule guarantees that the actor j can always use the latest data as computed by actor i, if the interaction ($i \longrightarrow j$) happens

4.1.3. Modeling actor concurrency in UPPAAL. In order to guarantee the correctness of temporal behavior of concurrently executing system actors, we need to properly handle the *discrete real-time* of COMDES-II systems in UPPAAL, which in principle executes with *continuous model-time*.

The real-time is the physical time elapsing autonomously in continuum in a constant manner[2], which can not be interrupted, frozen, or conserved for future use. Specifically, the real-time in COMDES-II is discretized as piece-wise points over continuum, whereby a piece is a basic timing unit equal to the execution period of the scheduler. Combining this timing policy with the timed multitasking model of computation implemented in COMDES-II, time is separated from functionality and specified with respect to actors as multiples of the basic timing unit: when the real-time reaches the activation time-stamp of an actor, the input drivers of the actor will be invoked with an assumption of zero logical execution time; when the real-time is equal to the deadline time-stamp of an actor, the output drivers of the actor will be executed to deliver output signals, logically in zero time too. Between these two time-stamps, there is no time information related to the actor's functionality. Under these assumptions, the temporal behavior of schedulable COMDES-II actors concerns only the timed I/O activities that are performed at precisely specified real-time stamps, which are of course denoted as multiples of the basic timing unit over continuum.

On the other hand, UPPAAL systems evolve with continuous model-time. The model-time is a kind of machine time having a polynomial relationship with the CPU oscillation speed, i.e. the model-time may elapse in continuum only when CPU oscillates. Within the closure of the physical world, the model-time does not proceed constantly: it may suffer from interrupts, preemptions from other computation processes, or it may feature frozen semantics (i.e. computation is carried on while time stops progressing). However, within the closure of the model world itself, the model-time indeed proceeds constantly: no matter how much real-time has actually elapsed, the time in model elapses with a constant pace anyhow. Aditionally, in UPPAAL clocks evolve synchronously in the real-value domain, but they can only capture the timing points at integer instants over continuum, which constitute conceptual *ticks* of clocks in the UPPAAL tool.

As a result, it is natural to map the basic timing unit in COMDES-II onto the conceptual tick of UPPAAL clocks, such that the temporal behavior of COMDES-II systems represented as multiples of the basic timing unit can be precisely specified in UPPAAL as multiples of the clock tick, since one time unit in COMDES-II is semantically equivalent to one (conceptual) clock tick in UPPAAL.

Based on this semantics, we model the COMDES-II actor concurrency in UPPAAL by adopting the following abstractions and assumptions.

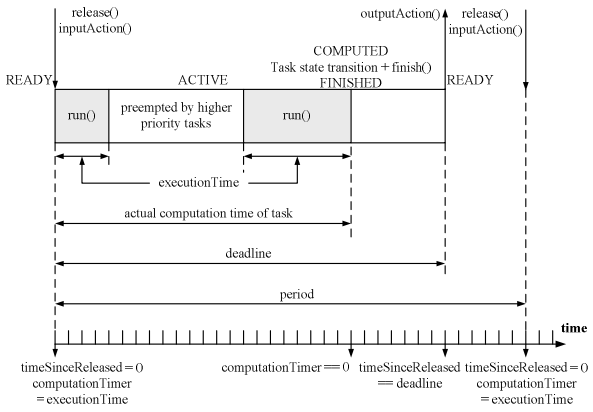

Figure 6. Actor concurrency in UPPAAL

An actor task may be conceptually in `READY`, `ACTIVE`, `COMPUTED` and `FINISHED` status if the actor is schedulable, as illustrated in Figure 6, otherwise the actor task will be in `ERROR` status (see Figure 5). Here, `READY` means that a task is ready for activation, and `ACTIVE` denotes that a task has been released, but not completed its computation yet. In a system it is always the highest-priority `ACTIVE` task to be running, whereby the corresponding task `computationTimer` is decremented with each invocation period of the scheduler. When the `computationTimer` of a specific `ACTIVE` task reduces to zero before its deadline, the task status will be set as `COMPUTED`, meaning that the computation effort has completed and the actor state transition may take place instantaneously, which is followed by the `FINISHED` status. The `FINISHED` status indicates that a particular task computation and control activities have already finished, such that the output signals are available for generation, when the as-

[2]Here we only consider the Newtonian notion of the physical time.

sociated deadline expires. If the `computationTimer` of an `ACTIVE` task is greater than zero (the task has not finished its computation) when its deadline comes, the task will be switched into the `ERROR` status (see Figure 5).

Manipulation of actor task execution status can be accomplished by invoking a number of scheduling primitives implemented in UPPAAL, including `release()`, `run()`, `finish()`, `outputAction()` and `inputAction()`. These primitives mimic their counterparts in COMDES-II, whose design philosophies are conceptually illustrated in Figure 6, and the implementations are referred to [10].

4.1.4. Modeling the discrete-time scheduler in UPPAAL. The approach to modeling COMDES-II actor concurrency using C-like programs in UPPAAL largely eases the effort to model the discrete-time scheduler, which is represented as a timed automaton shown in Figure 7.

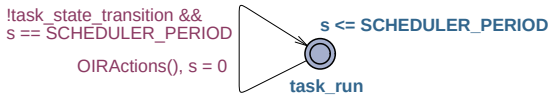

Figure 7. Discrete-time scheduler in UPPAAL

The scheduler contains only one location called `task_run`, and one edge guarded by Boolean conditions of `!task_state_transition && s == SCHEDULER_PERIOD`, in which:

- `s` is a clock variable that evolves autonomously in continuous time, whose value is confined by the `SCHEDULER_PERIOD` as an invariant condition specified in the `task_run` location
- `SCHEDULER_PERIOD` is an integer constant denoting the execution period of the scheduler, which is calculated as the *greatest common divisor* (GCD) of the non-zero `period`, `executionTime` and `deadline` of all actor tasks
- `task_state_transition` is a global Boolean variable used to resolve the transitions race problem, which may happen when both scheduler state transition and a task state transition can be fired simultaneously. The `task_state_transition` variable is set to `true` by a `COMPUTED` task to precede the state transition of the scheduler, and will be reset after the task state transition is `FINISHED`, such that the disabled scheduler execution is enabled again. In this way, non-deterministic concurrent state transitions are equivalently converted into deterministic sequential steps that execute logically in *zero* time, resulting in considerably reduced state space of verification

Two actions will be performed when the scheduler state transition takes place: `OIRActions()` and `s = 0`. Specifically, `OIRActions()` is a function invoking the `outputAction()`, `intputAction()` and `run()` primitives to execute the corresponding scheduling functionality in a sequential order. And `s = 0` resets the value of the clock, so that it is ready to count the time for the next cycle of scheduler execution.

4.2. Transformation of functionality

In COMDES-II the functional behavior of a system is described as a composition of different kinds of function blocks (FBs), which are intrinsically independent of the scheduling and timing issues specified at the actor level. Hence, system functionality can be directly transformed via equivalent FB models, regardless of the concurrency and timing information.

We model the COMDES-II basic, composite and modal FBs as functions handling integer variables or data structures in the transformed UPPAAL models. In particular, the basic and/or composite FBs preprocessing the event/guard signals for a state machine FB can be implemented as UPPAAL functions, which are invoked in the `taskInputDrivers(int taskID)` primitive when the host actor is activated. The modal FBs may be treated in a similar way but the corresponding functions will be executed in the `taskOutputDrivers(int taskID)` primitive when the host actor deadline expires. To this end, the *state* and *state_updated* input variables required by a modal FB (see Figure 3) can be obtained from the `mode` and `modeUpdated` entries of the task control block instance indexed by `taskID`.

On the other hand, system reactive behavior specified by state machine FBs can be transformed into equivalent timed automata in UPPAAL *without* timing annotations, since time is not involved in the functionality. An example automaton is given in Figure 8, whose semantics is equivalent to the *SMFB_1* state machine FB shown in Figure 4. The model contains also two locations *s1* and *s2*, and a number of edges tagged with guards and update actions.

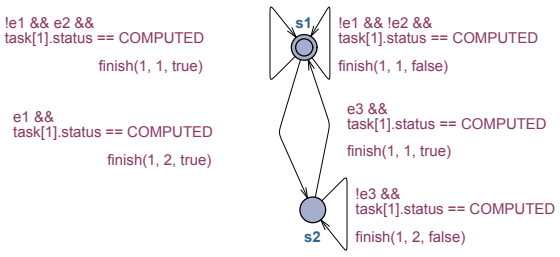

Figure 8. An UPPAAL automaton for *SMFB_1*

A guard basically consists of two parts: one is the application-specific event, e.g. *e1* in the edge from *s1* to *s2*. The other one is a condition expression justifying if the status of the host actor task is `COMPUTED` or not, as explained in Section 4.1.3. For instance, the condition expression `task[1].status == COMPUTED` is labeled in all edges, in which `task[]` is an array of instances of the task control block, and `1` indicates the task priority. Once a transition has taken place, the `finish()` primitive will be invoked to update the `mode` and `modeUpdated` entries associated with the corresponding task, and then change the

task status to FINISHED (see Figure 6).

In COMDES-II, all state transitions of a state machine FB are ordered, such that a higher-order transition from a source state will preempt a lower-order transition from the same source state, if both of them can be fired. This order(priority)-based precedence can also be realized via mutually exclusive transition triggers, whereby the explicit orders become unnecessary. Specifically, the trigger of a lower-order transition should be complemented with the negation of all higher-order transition triggers, meaning that the lower-order transition can only be fired when its trigger is present while all higher-order transition triggers are absent. This mechanism of imposing preemption on state transitions is adopted in the UPPAAL model of state machine FBs, as exemplified in Figure 8.

The presented model transformation method has been practically validated via a Turntable Control System case study [11, 12], which was first designed in COMDES-II, and subsequently transformed into UPPAAL for verification against a set of system requirements. The verification results are promising in the sense that not only the system behaves as expected, but also the verification process of each property requires a modest amount of resources. For example, the system schedulability can be checked against a temporal logic property expressed as:

```
A[] forall(i :  int[1,TASKS_NUM])
task[i].status != ERROR,
```

which means that along all possible execution traces (A) and in all reachable states ([]) any task can not be in an ERROR status, where TASKS_NUM represents the number of actor tasks (6 in this case) in the UPPAAL model. The checking was performed within 7s on a computer having 2.0 GHz Core Duo CPUs and 2 GBytes RAM, with 18220 Kbytes of memory footprint. A full description of the verification effort of the Turntable Control System in UPPAAL is referred to [10].

5. Conclusion

The paper has investigated a transformational approach to the formal verification of dynamic behavior of COMDES-II systems using UPPAAL. The main contribution of this paper lies in the fact that the proposed approach allows for precise analysis of *schedulability* and *reactive behavior* of timed multitasking systems, in the context of sequential control applications operating in the discrete real-time domain.

The adopted methodology of semantic anchoring provides a theoretical foundation for the model transformation that equivalently anchors the behavioral semantics of COMDES-II onto UPPAAL timed automata at the meta-level. The developed model transformation technique has been used to verify a practical case study designed in COMDES-II – the Turntable Control System – against a list of desired system requirements. The verification results demonstrate that the transformed system can be effectively analyzed with a complete preservation of the original system semantics.

References

[1] Chen, J. Sztipanovits and S. Neema, "Toward a semantic anchoring infrastructure for domain-specific modeling languages," in *Proceedings of the 5th ACM international conference on Embedded software*, Jersey City, NJ, USA, 2005.

[2] J. Reekie and E. A. Lee, "Lightweight Component Models for Embedded Systems," U.C. Berkeley, CA 94720, USA, Technical Memorandum UCB/ERL M02/30, October 30, 2002.

[3] C. Angelov, Xu Ke and K. Sierszecki, "A Component-Based Framework for Distributed Control Systems," in *Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2006)*, Cavtat-Dubrovnik, Croatia, August, 2006.

[4] Xu Ke, K. Sierszecki, C. Angelov, "COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems," in *Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Daegu, S.Korea, 2007.

[5] A. Narayanan and G. Karsai, "Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations," in *Proceedings of the Second International Workshop on Graph and Model Transformation*, Brighton, UK, 2006.

[6] C. Angelov and J. Berthing, "Distributed Timed Multitasking: a Model of Computation for Hard Real-Time Distributed Systems," in *Proc. of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems DIPES'06*, Braga, Portugal, Oct. 2006.

[7] G. Behrmann, A. David, and Kim G. Larsen, "A Tutorial on UPPAAL," in *Proc. of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. LNCS 3185.

[8] Kim G. Larsen, Paul Pettersson and Wang Yi, "Uppaal in a Nutshell," *Springer International Journal of Software Tools for Technology Transfer*, vol. 1+2, 1997.

[9] Robin Milner, *Communication and Concurrency*. Prentice Hall, 1989, ISBN 0-13-114984-9.

[10] Xu Ke, P. Pettersson, K. Sierszecki and C. Angelov, "Verification of COMDES-II Systems Using UPPAAL with Model Transformation," Mads Clausen Intitute, University of Southern Denmark, Soenderborg, Denmark, Technical report, Oct. 2007.

[11] Bos, V., Kleijn, J.J.T., "Automatic verification of a manufacturing system," *Robotics and Computer Integrated Manufacturing*, vol. 17, pp. 185–198, 2001.

[12] Bortnik, E., Trcka, N., Wijsc, A.J., Luttik, B., van de Mortel-Fronczak, J.M., Baeten, J.C.M., Fokkink, W.J., Rooda, J.E., "Analyzing a gamma Model of a Turntable System using Spin, CADP and Uppaal," *Journal of Logic and Algebraic Programming*, vol. 65, pp. 51–104, 2005.